Gap Puzzles

Puzzle solving using Prolog constraints

João Pedro Pinheiro de Lacerda Campos[1,2] and Leonardo Fernandes Moura[1,2]

[1] Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, s/n
4200-465 Porto PORTUGAL `feup@fe.up.pt`
http://www.fe.up.pt

[2] Logical Programing Course, Class 3MIEIC03, Group Gap Puzzle_5

Abstract. This project was developed during the the Logical Programing Course in the context of the year of the Integrated Masters in in Informatics and Computing Engineering at The Faculty of Engineering of the University of Porto. The goal was to solve a problem using the Prolog programming language, more specifically, using Constraint Logic Programming over Finite Domains (clpfd) library, that's defined in the SICStus Prolog distribution. In the specific case of our team, the problem in question is the Gap Puzzle, which is defined in detail in **Section 2**. In order to solve the problem, Prolog constraint programming was used. This type of programming grants us the advantage of fast computation, requiring a somewhat simple and little verbosed kind of programming. Using constraints, we can simply define all the constraints of our problem (in the case of puzzles these are the rules of the puzzle), which in the case of more complex problems is not trivial. Then, after defining the solution search algorithm, the clpfd library will search for solutions. **CONCLUSÕES**

Keywords: Prolog · Constraint Programing · Gap Puzzle.

1 Introduction

This goal of the project was to elaborate a Prolog program that, given a board order, computes all the solutions of the Gap Puzzle for that board order. Optimization is a relevant factor here. Different ways of defining constraints can lead us to the same set of solutions, but computation times can vary greatly. So, one of the concerns during the development of this project was to find solutions as fast as possible.

Bellow we shortly explain the following sections of this article.

- **Problem Description:** Detailed description of the problem we are solving and the rules of Gap Puzzles
- **Approach:**
  - **Decision Variables:**
  - **Constraints:**
  - **Search Strategy:**
- **Solution Presentation:** Description of how solutions are presented in an intelligible way.
- **Results:**
- **Conclusions:**

2 Problem Description

The problem in question is the solution is the solution of the Gap Puzzle. Gap is a very simple puzzle, yet a challenging one. We have grid of arbitrary size and our goal, is to shade two squares of the grid in every row and column, such that two shaded squares do not touch, even at the corners. Because there is more than one solution for every grid order, in the puzzle form of the problem, there are numbers on the side of the grid to indicate the number of non shaded squares there must exist between two shaded squares in that row or column. The figure below illustrates an example of puzzle with order 5.

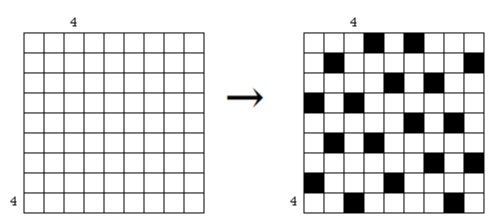

Fig. 1. Gap Puzzle example

Although they are very interesting, these specific puzzles do not interest us. We are interested in finding arbitrary solutions for a problem with a given order.

3 Approach

In the specific context of this problem, the constraints used to solve the problem are a translation of the rules of the puzzle to Prolog Constraints. The following subsections explain the constraints used in detail.

3.1 Decision Variables

To solve this problem we have only a decision variable, the *Points* list. This list stores the coordinates of all the cells that will be shaded, in the format *[X1,Y1,X2,Y2—...]*. We start by defining that this list will have a a length of:

$$points = 2 * order \tag{1}$$

$$length = 4 * order \tag{2}$$

This happens because we can only have two shaded cells in each row. Since the number of rows is equal to the order of our grid, we a number of points that is two times our order.

Then we define the domain of the Decision Variable, this will between zero and the number immediately inferior to our order, this is because we are considering a system of coordinates starting in zero.

3.2 Constraints

The first constraint we make is to grant there are only two shaded cells in each row and in each column. We do this using the **restrict_two(+List,+Order)** predicate. This predicate restricts each number of the domain in the list to occur exactly two times in the list. We do this two times, the first for a list containing all the X coordinates of the solution, and a second time for the Y coordinate. We are able to split this using the **extractX(+List,-XList)** and **extractY(+List,-YList)** predicates. The first one takes a list with the solution format and unifies in *XList* all the X coordinates of that list. The *extractY* predicate does the same for Y.

```
restrict_two(Sol,0) :- count(0,Sol,#=,2).

restrict_two(Sol,S) :-
    count(S,Sol,#=,2),
    Next is S-1,
    restrict_two(Sol,Next).
```

Fig. 2. The restrict_two predicate

Then we restrict the positions of the shaded cells themselves. We do this using the **restrict_next(+List)** predicate. This predicate iterates through the list making sure that no other coordinates in the list are the coordinates of an adjacent cell to the any of the other cells and that no two cell coordinates are the same. This predicate calls the **restrict_next_pair(+List,+X,+Y)** predicate that applies the constraints mentioned above to all the other coordinate pairs in the list.

```
restrict_next_pair([],_,_).

restrict_next_pair([HX,HY|T],X,Y) :-
    #\((HX #= X + 1) #/\ (HY #= Y + 1)),
    #\((HX #= X + 1) #/\ (HY #= Y - 1)),
    #\((HX #= X - 1) #/\ (HY #= Y + 1)),
    #\((HX #= X - 1) #/\ (HY #= Y - 1)),
    #\((HX #= X + 1) #/\ (HY #= Y)),
    #\((HX #= X - 1) #/\ (HY #= Y)),
    #\((HX #= X) #/\ (HY #= Y - 1)),
    #\((HX #= X) #/\ (HY #= Y + 1)),
    #\((HX #= X) #/\ (HY #= Y)),
    restrict_next_pair(T,X,Y).
```

Fig. 3. The restrict_next_pair predicate

4 Search Strategy

So that prolog can search for arbitrary solutions and not different permutations of the same solution, we need to define a predicate that makes each solution unique. In the present case, this was accomplished by ordering all the points of a solution in a crescent order, making each solution unique. This is accomplished by the **crescent_order(+List)** predicate.

```
crescent_order([_,_]).

crescent_order([X1,Y1,X2,Y2|T]) :-
    X1 #>= X2,
    #\((X1 #= X2) #<=> (Y1 #> Y2)),
    crescent_order([X2,Y2|T]).

crescent_order([H1,H2|T]) :-
    H1 #>= H2,
    crescent_order([H2|T]).
```

Fig. 4. The crescent_order predicate

5 Solution Presentation

For solution presentation there the **display_gap(+Sol,+Len)** was elaborated. This predicate receives the *Sol* list, which is a prolog list containing the coordinates of all grid cells to be shaded for that solution, and *Len*, which is the length of a row, that is, the order of the board the solution given applies to. In order to do this, this predicate uses the three predicates explained bellow.

This predicate uses the **empty_board(-B,+Len)** which creates a list of lists representing an empty grid, with order *Len*, this new grid is then unified with *B*. This predicate works by appending new atoms, representing an empty cell (**' '**, a single space character) to an empty starting list. Once this list has the desired length, this list is then replicated several times to form a new empty grid.

Then, we use the **process_board(+Board,+Sol,-NewBoard)** to substitute all the empty cells in the positions given by solution list, with the **'X'** atom. This represents a shaded cell. In order to make the substitution we defined the **replaceN(+List,+N,+NewElem,Res)** predicate. This predicate substitutes the element of order *N* in *List* to the element *NewElem*, then unifies the resulting list with *NewBoard*.

Finally, the predicate **display_board(+Board)** was defined to display the board *Board*. This predicate writes the separators in the screen and then displays each row of the grid, using the right format. Bellow, we can see a displayed board.

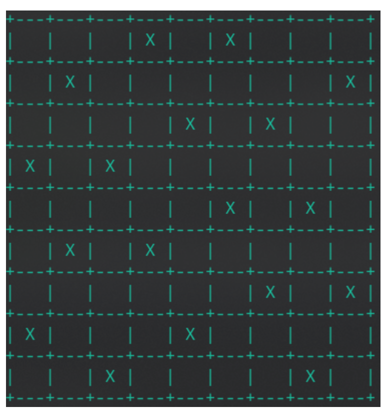

Fig. 5. Solution Display Example

6 Results

7 Conclusions and Future Work

References

1. Erich Friedman's Puzzle Palace - Gap Puzzles, Copyright Erich Friedman, 2010. https://www2.stetson.edu/ efriedma/puzzle/gap/. Last accessed 4 Jan 2020

8 Introduction

8.1 A Subsection Sample

Please note that the first paragraph of a section or subsection is not indented. The first paragraph that follows a table, figure, equation etc. does not need an indent, either.

Subsequent paragraphs, however, are indented.

Sample Heading (Third Level) Only two levels of headings should be numbered. Lower level headings remain unnumbered; they are formatted as run-in headings.

Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Table 1 gives a summary of all heading levels.

Table 1. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	**Lecture Notes**	14 point, bold
1st-level heading	**1 Introduction**	12 point, bold
2nd-level heading	**2.1 Printing Area**	10 point, bold
3rd-level heading	**Run-in Heading in Bold.** Text follows	10 point, bold
4th-level heading	*Lowest Level Heading.* Text follows	10 point, italic

Displayed equations are centered and set on a separate line.

$$x + y = z \tag{3}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. 6).

Theorem 1. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

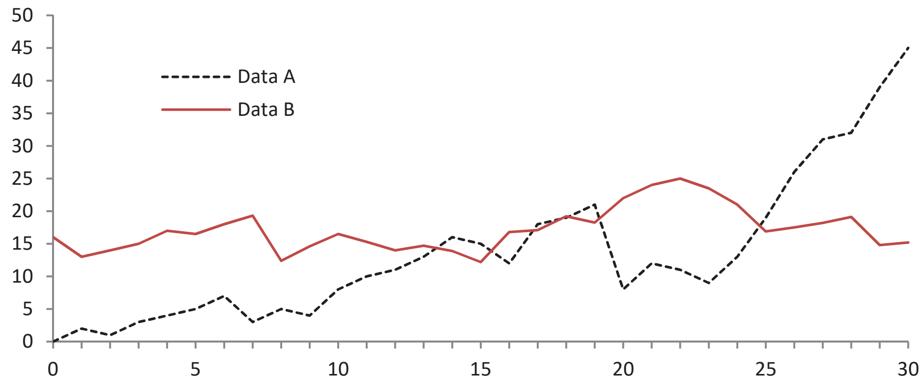

Fig. 6. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [1], an LNCS chapter [2], a book [3], proceedings without editors [4], and a homepage [5]. Multiple citations are grouped [1–3], [1, 3–5].

References

1. Author, F.: Article title. Journal **2**(5), 99–110 (2016)
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016). https://doi.org/10.10007/1234567890
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999)
4. Author, A.-B.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010)
5. LNCS Homepage, http://www.springer.com/lncs. Last accessed 4 Oct 2017